Пензенский государственный университет

Кафедра «Вычислительная техника»

**ОТЧЕТ**

по лабораторной работе №6

# по курсу «Логика и основы алгоритмизации в инженерных задачах»

на тему «Поиск расстояний в графе»

**Выполнили:**

студенты группы 20ВВ3

Шмелёв Д.В.

**Приняли:**

Юрова О.В.

Митрохин М.А.

Пенза 2021

# Название

Поиск расстояний в графе

**Цель работы** – изучение алгоритма поиска расстояний в графе.

# Методические указания

Поиск расстояний – довольно распространенная задача анализа графов.

Для поиска расстояний можно использовать процедуры обхода графа.

Для этого при каждом переходе в новую вершину необходимо запоминать, сколько шагов до нее мы сделали. При этом вектор, который хранил информацию о посещении вершин становится вектором расстояний.

# Лабораторное задание

**Задание 1**

1. Сгенерируйте (используя генератор случайных чисел) матрицу

смежности для неориентированного графа G. Выведите матрицу на экран.

2. Для сгенерированного графа осуществите процедуру поиска

расстояний, реализованную в соответствии с приведенным выше описанием.

При реализации алгоритма в качестве очереди используйте класс queue из

стандартной библиотеки С++.

3.\* Реализуйте процедуру поиска расстояний для графа, представленного

списками смежности.

**Задание 2\***

1. Реализуйте процедуру поиска расстояний на основе обхода в глубину.

2. Реализуйте процедуру поиска расстояний на основе обхода в глубину

для графа, представленного списками смежности.

3. Оцените время работы реализаций алгоритмов поиска расстояний на

основе обхода в глубину и обхода в ширину для графов разных порядков.

**Псевдокод**

Вход: G – матрица смежности графа, v – исходная вершина.

Выход: DIST – вектор расстояний до всех вершин от исходной.

Алгоритм ПОШ

1.1. для всех i положим DIST [i] = -1 пометим как &quot;не посещенную&quot;;

1.2. ВЫПОЛНЯТЬ BFSD (v).

1.3 для всех i вывести DIST [i] на экран;

Алгоритм BFSD(v):

2.1. Создать пустую очередь Q = {};

2.2. Поместить v в очередь Q.push(v);

2.3. Обновить вектор расстояний DIST [ x ] = 0;

2.4. ПОКА Q != Æ очередь не пуста ВЫПОЛНЯТЬ

2.5. v = Q.front() установить текущую вершину;

2.6. Удалить первый элемент из очереди Q.pop();

2.7. вывести на экран v;

2.8. ДЛЯ i = 1 ДО size\_G ВЫПОЛНЯТЬ

2.9. ЕСЛИ G(v,i) = = 1И DIST = = -1

2.10. ТО

2.11. Поместить i в очередь Q.push(i);

2.12. Обновить вектор расстояний DIST [ i ] = DIST [ v ] + 1;

# Листинг

#define \_CRT\_NONSTDC\_NO\_WARNINGS

#define \_CRT\_SECURE\_NO\_WARNINGS

#define HEADER ("Лабораторная работа №6\nВыполнили: Шмелёв Д. и Пантюшов Е.\n\n")

#include <stdio.h>

#include <locale.h>

#include <stdlib.h>

#include <time.h>

#include <queue>

#include <stack>

using namespace std;

struct node

{

int vertex;

struct node\* next;

};

struct Graph

{

int numVertices;

struct node\*\* adjList;

};

struct node\* createNode(int v)

{

struct node\* newNode = (node\*)(malloc(sizeof(struct node)));

newNode->vertex = v;

newNode->next = NULL;

return newNode;

}

void addEdge(struct Graph\* graph, int i, int j)

{

struct node\* newNode = createNode(j);

if (graph->adjList[i] == 0)

{

graph->adjList[i] = newNode;

newNode = NULL;

}

struct node\* temp = graph->adjList[i];

while (temp->next)

{

temp = temp->next;

}

temp->next = newNode;

newNode = createNode(i);

if (graph->adjList[j] == 0)

{

graph->adjList[j] = newNode;

newNode = NULL;

}

temp = graph->adjList[j];

while (temp->next)

{

temp = temp->next;

}

temp->next = newNode;

}

struct Graph\* createGraph(int\*\* M, int vertices)

{

struct Graph\* G = (Graph\*)(malloc(sizeof(struct Graph)));

G->numVertices = vertices;

G->adjList = (node\*\*)(malloc(vertices \* sizeof(struct node\*)));

for (int i = 0; i < vertices; i++)

G->adjList[i] = NULL;

for (int i = 0; i < vertices; i++)

{

for (int j = 0; j < vertices; j++)

{

if ((M[i][j] == 1) && (i < j))

{

addEdge(G, i, j);

}

}

}

return G;

}

void printGraph(struct Graph\* graph)

{

for (int i = 0; i < graph->numVertices; i++)

{

struct node\* temp = graph->adjList[i];

printf("%d-я вершина: ", i);

while (temp)

{

printf("%d ", temp->vertex);

temp = temp->next;

}

printf("\n");

}

}

void dfsd\_list(struct Graph\* G, int n, int\* dist, int\* NUM, int x)

{

stack <int> s;

struct node\* v = G->adjList[x];

s.push(x);

dist[x] = 0;

while (!s.empty())

{

x = s.top();

s.pop();

v = G->adjList[x];

if (NUM[x] == 2)

continue;

NUM[x] = 2;

while(v)

{

if (NUM[v->vertex] != 2 && dist[v->vertex] == -1)

{

s.push(v->vertex);

NUM[v->vertex] = 1;

dist[v->vertex] = dist[x] + 1;

}

v = v->next;

}

}

}

void DFSD\_LIST(struct Graph\* G, int n, int\* dist, int x, int k)

{

struct node\* v = G->adjList[x];

int r = k + 1;

dist[x] = k;

while(v)

{

if (dist[v->vertex] == -1)

DFSD\_LIST(G, n, dist, v->vertex, r);

if (k < dist[v->vertex])

DFSD\_LIST(G, n, dist, v->vertex, r);

v = v->next;

}

}

void DFSD(int\*\* M, int n, int\* dist, int x, int k)

{

int r = k + 1;

dist[x] = k;

for (int i = 0; i < n; i++)

{

if (M[x][i] == 1 && dist[i] == -1)

DFSD(M, n, dist, i, r);

if (M[x][i] == 1 && k < dist[i])

DFSD(M, n, dist, i, r);

}

}

void dfsd(int\*\* M, int n, int\* dist, int\* NUM, int x)

{

stack <int> s;

s.push(x);

dist[x] = 0;

while (!s.empty())

{

x = s.top();

s.pop();

if (NUM[x] == 2)

continue;

NUM[x] = 2;

for (int i = n - 1; i > -1; i--)

{

if (M[x][i] == 1 && NUM[i] != 2 && dist[i] == -1)

{

s.push(i);

NUM[i] = 1;

dist[i] = dist[x] + 1;

}

}

printf(" %d \n", x);

}

}

void bfsd\_list(struct Graph\* G, int n, int\* dist, int x)

{

queue <int> q;

struct node\* v = G->adjList[x];

q.push(x);

dist[x] = 0;

while (!q.empty())

{

x = q.front();

v = G->adjList[x];

q.pop();

while (v)

{

if (dist[v->vertex] == -1)

{

q.push(v->vertex);

dist[v->vertex] = dist[x] + 1;

}

v = v->next;

}

}

}

void bfsd(int\*\* M, int n, int\* dist, int x)

{

queue <int> q;

q.push(x);

dist[x] = 0;

while (!q.empty())

{

x = q.front();

q.pop();

for (int i = 0; i < n; i++)

{

if (M[x][i] == 1 && dist[i] == -1)

{

q.push(i);

dist[i] = dist[x] + 1;

}

}

printf(" %d \n", x);

}

}

void f5(struct Graph\* G, int n)

{

int x, k = 0;

int\* dist = (int\*)(malloc(n \* sizeof(int)));

//int\* NUM = (int\*)(malloc(n \* sizeof(int)));

printf("\n\nПункт 2.\nВведите точку входа: ");

scanf("%d", &x);

for (int i = 0; i < n; i++)

{

dist[i] = -1;

}

DFSD\_LIST(G, n, dist, x, k);

for (int i = 0; i < n; i++)

{

if (dist[i] == -1)

{

dist[i] = 0;

}

printf("%d -> %d: %d\n", x, i, dist[i]);

}

}

void f4(int\*\* M, int n)

{

int x, k = 0;;

int\* dist = (int\*)(malloc(n \* sizeof(int)));

//int\* NUM = (int\*)(malloc(n \* sizeof(int)));

printf("\nЗадание 2.\n\nПункт 1.\nВведите точку входа: ");

scanf("%d", &x);

for (int i = 0; i < n; i++)

{

dist[i] = -1;

}

DFSD(M, n, dist, x, k);

for (int i = 0; i < n; i++)

{

if (dist[i] == -1)

{

dist[i] = 0;

}

printf("%d -> %d: %d\n", x, i, dist[i]);

}

}

void f3(int\*\* M, struct Graph\* G, int n)

{

int x;

int\* dist = (int\*)(malloc(n \* sizeof(int)));

printf("\n\nПункт 3.\nСписок смежности:\n");

printGraph(G);

printf("\nВведите точку входа: ");

scanf("%d", &x);

for (int i = 0; i < n; i++)

{

dist[i] = -1;

}

bfsd\_list(G, n, dist, x);

for (int i = 0; i < n; i++)

{

if (dist[i] == -1)

{

dist[i] = 0;

}

printf("%d -> %d: %d\n", x, i, dist[i]);

}

}

void f2(int\*\* M, int n)

{

int x;

int\* dist = (int\*)(malloc(n \* sizeof(int)));

printf("\nПункт 2.\nВведите точку входа: ");

scanf("%d", &x);

for (int i = 0; i < n; i++)

{

dist[i] = -1;

}

bfsd(M, n, dist, x);

for (int i = 0; i < n; i++)

{

if (dist[i] == -1)

{

dist[i] = 0;

}

printf("%d -> %d: %d\n", x, i, dist[i]);

}

}

void f1(int\*\* M, int n)

{

printf("\nЗадание 1.\n\nПункт 1.\nМатрица смежности:\n");

for (int i = 0; i < n; i++)

{

M[i] = (int\*)(malloc(n \* sizeof(int)));

for (int j = 0; j < n; j++)

{

M[i][j] = rand() % 100;

if (M[i][j] < 40)

{

M[i][j] = 0;

}

else

{

M[i][j] = 1;

}

}

}

for (int i = 0; i < n; i++)

{

for (int j = 0; j < n; j++)

{

if (i == j)

{

M[i][j] = 0;

}

else

{

M[i][j] = M[j][i];

}

printf(" %d ", M[i][j]);

}

printf("\n");

}

}

int main()

{

setlocale(LC\_ALL, "RUS");

srand(time(NULL));

printf(HEADER);

int n;

printf("Введите размерность матрицы: ");

scanf("%d", &n);

int\*\* M = (int\*\*)(malloc(n \* sizeof(int\*)));

f1(M, n);

clock\_t t1 = clock();

f2(M, n);

clock\_t t2 = clock();

float t3 = t2 - t1;

printf("Время: %f", t3 / 1000);

struct Graph\* G = createGraph(M, n);

f3(M, G, n);

t1 = clock();

f4(M, n);

t2 = clock();

t3 = t2 - t1;

printf("Время: %f", t3 / 1000);

f5(G, n);

system("PAUSE");

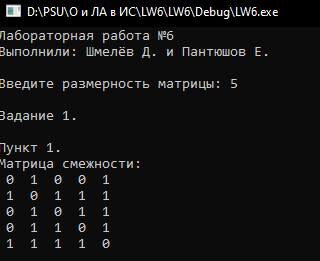
return 0;

# }

# Результат работы программы

**Пункт 1.1.**

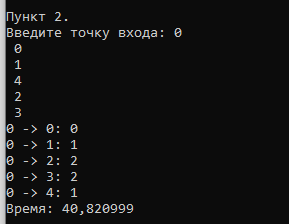
Результаты работы программы показаны на рисунке 1.



**Рисунок 1 – Результат работы программы**

**Пункт 1.2.**

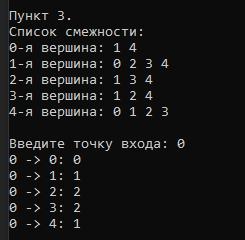
Результаты работы программы показаны на рисунке 2.



**Рисунок 2 – Результат работы программы**

**Пункт 1.3.**

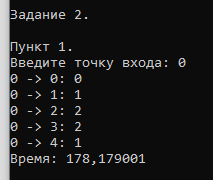
Результаты работы программы показаны на рисунке 3.



**Рисунок 3 - Результат работы программы**

**Пункт 2.1.**

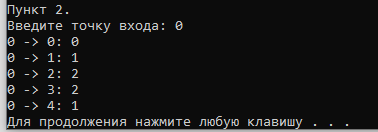
Результаты работы программы показаны на рисунке 4.

****

**Рисунок 4 – Результат работы программы**

**Пункт 2.2.**

Результаты работы программы показаны на рисунке 5.

****

**Рисунок 5 – Результат работы программы**

# Вывод

# В ходе выполнения лабораторной работы была разработана программа, в который были реализованы алгоритмы поиска расстояний в графе, используя поиски в ширину и глубину. При разработке данного алгоритма были применены такие структуры данных как очередь и стек.